Chapter 5

The K Semantic Framework

This chapter introduces K, a rewriting-based semantic definitional framework for programming languages and calculi, which will be used in the remainder of this book. The K framework consists of two components:

1. The *K concurrent rewrite abstract machine*, abbreviated KRAM and discussed in Section 5.3;
2. The *K technique*, discussed in Section 5.4.

Like in term rewriting, a K system consists of a signature for buinding terms and of a set of rules for iteratively rewriting terms. Like in rewriting logic [22], K rules can be applied concurrently and unrestricted by context. The novelty of the KRAM (Section 5.3) is that its rules contain, besides expected information saying how the original term is modified (the *write data*), also information about what parts of the term are shared with other rules (the *read-only data*) as well as what parts of the term can be concurrently modified by other rules (the *volatile data*). This additional information allows K to be a trully concurrent definitional framework. Its concurrent rewrite steps may not always be serialized, which means that they cannot necessarily be simulated (interleaved) using conventional term rewriting. Special conditions are given which ensure serializability of K concurrent rewrites; however, even in this case, the K concurrent rewrites associated to a K system may require several interleaved rewrites in the rewrite logic theory straightforwardly (i.e., by forgetting the read-only and the volatile data information) associated to the K system.

The KRAM aims at maximizing the amount of concurrency that can be achieved in a rewriting setting, so that it can serve as an underlying infrastructure for giving semantics to trully concurrent programming languages. What KRAM does not do is to tell *how* one can define a programming language or a calculus as a K system. In particular, a bad K definition may partially or totally inhibit KRAM's potential for concurrency. The K technique discussed in Section 5.4 proposes an approach and supporting notation to make the use of the KRAM convenient when formally defining programming languages and calculi. Moreover, if one does not make use of the non-serializable aspects of the KRAM, the K technique can also be and actually has already been intensively used as a technique to define languages and calculi as term rewrite systems or as rewrite logic theories. There is one important thing lost in the translation of the K notation into rewriting logic though, namely the degree of true concurrency of the original K definition. Ignoring this true concurrency aspect, the relationship between "serializable" K and rewriting logic in general and Maude in particular is the same as that between any of the conventional semantic styles discussed in Chapter 3 and rewriting logic and Maude: the latter can be used to execute and analyze K definitions.

5.1 Quest for an Ideal Language Definitional Framework

Chapter 3 showed that any conventional language definitional style can be faithfully, step-for-step, captured by a rewriting logic theory. It may then seem "obvious" to the hasty reader that rewriting logic is perhaps the ideal language definitional framework and thus naturally ask the following:

> *What is the need for yet another language definitional framework that can be embedded in rewriting logic, K in this case, if rewriting logic is already so powerful?*

Unfortunately, in spite of its definitional strength as a computational logic framework, rewriting logic does not give, and does not intend to give, the programming language designer any recipe on *how* to define a language. It essentially only suggests the following: however one wants to formally define a programming language or calculus, one can probably also do it in rewriting logic following the same intuitions and style. Therefore, rewriting logic can be regarded as a *meta-framework* that supports definitions of programming languages and calculi among many other things, providing the language designer with a means to execute and formally analyze languages in a generic way, but only *after* the language is already defined. The following important (multi-)question remains largely open to the working programming language designer, and not only:

> *Is there any language definitional framework that, at the same time,*
>
> 1. *Gives a strong intuition, even precise recipes, on how to define a language?*
> 2. *Same for language-related definitions, such as type checkers, type inferencers, abstract interpreters, safety policy or domain-specific checkers, etc.?*
> 3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*
> 4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*
> 5. *Supports non-determinism and concurrency, at any desired granularity?*
> 6. *Is generic, that is, not tied to any particular programming language or paradigm?*
> 7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*
> 8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*
> 9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

The list above contains a *minimal* set of desirable features that an ideal language definitional framework should have. Unfortunately, the current practice is to take the above features one at a time, temporarily or permanently declaring the others as "something else". We next describe how current practices and language definitional styles fail to satisfy the above-mentioned requirements.

1. *Gives a strong intuition, even precise recipes, on how to define a language?*

   To formalize one's intuition about a language feature, it is common practice to use a big-step or a small-step SOS definition, with or without evaluation contexts, typically on paper, without

any machine support. Sometimes this so-called "formal" process is pushed to extreme in what regards its informality, in the sense that one can see definitions of some language features using one definitional style and of other features using another definitional style, without ever proving that the two definitional styles can co-exist in the claimed form for the particular language under consideration. For example, one may use a big-step SOS to give semantics to a code-self-generation extension of Java, while using a small-step SOS to define the concurrency semantics of Java. However, once one has concurrency and shared memory, one cannot have a big-step SOS definition. An ideal language definitional framework should provide a uniform, compact and rigorous way to modularly define various language features, avoiding the need to define different language features following different styles.

2. *Same for language-related definitions, such as type checkers, type inferences, abstract interpreters, safety policy or domain-specific checkers, etc.?*

   To define a type system or a (domain-specific or not) safety policy for a language, one may follow a big-step-like definitional style, or even simply provide an algorithm to serve as a formal definition. While this appears to be, and in many cases indeed is acceptable, there can be a significant "formal gap" between the actual language semantic definition and its type system or safety policy regarded as mathematical objects, because in order to carry out proofs relating the two one needs one common formal ground. In practice, one typically ends up "encoding" the two in yet another framework, claimed to be "richer", and then carry out the proofs within that framework. But how can one make sure that the encodings are correct? Do they serve as alternative definitions for that sole purpose?

   An ideal language definitional framework should have all the benefits of the "richer" framework, at no additional notational or logical complexity, yet naturally capturing the complete meaning of the defined constructs. In other words, in an ideal framework one should define a language as a mathematical object, say $\mathcal{L}$, and a type system or other abstract interpretation of it as another mathematical object over the same formalism, say $\mathcal{L}'$, and then carry out proofs relating $\mathcal{L}$ and $\mathcal{L}'$ using the provided proof system of the definitional framework. $\mathcal{L}$, $\mathcal{L}'$, as well as other related definitions, should be human readable and easy to understand enough so that one does not feel the drive to give alternative, more intuitive definitions using a more informal notation or framework.

3. *Can define arbitrarily complex language features, including, obviously, all those found in existing languages, capturing also their intended computational granularity?*

   Some popular language definitional frameworks are incapable of defining even existing language features. The fact that a particular language feature is supported in some existing language serves as the strongest argument that that feature may be desirable, so an ideal language definitional framework must simply support it; in other words, one cannot argue against the usefulness of that feature just because one's favorite definitional framework does not support it. For example, since in standard SOS definitions (not including reduction semantics with evaluation contexts) the "control flow" information of a program is captured within the structure of the "proof", and since proof derivations are not first class objects in these formalisms, it makes it very hard, virtually impossible in these formalisms to define complex control intensive language constructs like, e.g., call-with-current-continuation (callcc).

   Another important example showing that conventional definitional frameworks (e.g., SOS)

fail to properly support existing common language features, is concurrency. Most frameworks enforce an interleaving semantics, which may not necessarily always be the desired approach to concurrency. In particular, an implementation of a multi-threaded system in which two threads can concurrently read a shared variable would be dissallowed, because it disobeys the "formal" interleaving-based language definition. Concurrency is further discussed in item 5.

Some frameworks provide a "well-chosen" set of constructs, shown to be theoretically sufficient to define any computable function or algorithm, and then propose *encodings* of other language features into the set of basic ones; examples in this category are Turing machines or the plethora of (typed or untyped) λ-calculi, or π-calculi, etc. While these basic constructs yield interesting and meaningful idealized programming languages, using them to encode other language features is, in our view, inappropriate. Indeed, encodings hide the intended *computational granularity* of the defined language constructs; for example, a variable lookup intended to be a one-step operation in one's language should take precisely one step in an ideal framework (not hundreds/thousands of steps as in a Turing machine or lambda calculus encoding, not even two steps: first get location, then get value).

4. *Is modular, that is, adding new language features does not require to modify existing definitions of unrelated features? Modularity is crucial for scalability and reuse.*

   As Mosses pointed out in [33] and as shown in Chapter 3 and discussed in Section 3.9, big-step and small-step SOS are non-modular; Plotkin himself had to modify the definition of simple arithmetic expressions (in the original notes on SOS [35]) three times as his initial language evolved. As seen in Section 3.8, to add an innocent abrupt termination statement to a language defined using SOS, say a `halt`, one needs to more than double the total number of rules: each language construct needs to be allowed to "propagate" the halting signal potentially generated by its arguments. Also, as one needs to add more items into configurations to support new language features, in SOS one needs to change again every rule to include the new items; note that there are no less than $7 + n * 10$ configuration items, where n is the number of threads, in the configuration of SKOOL in Section B.4 (which is a comparatively simple language) as shown in Figure 5.6. It can easily become very annoying and error prone to modify a large portion of unrelated existing definitions when adding a new feature.

   A language designer may be unwilling to add a new feature or improve the definition of an existing one, just because of the large number of required changes. Informal writing conventions are sometimes adopted to circumvent the non-modularity of SOS. For example, in the definition of Standard ML [31], Milner and his collaborators propose a "store convention" to avoid having to mention the store in every rule, and an "exception convention" to avoid having to double the number of rules for the sole purpose of supporting exceptions. As rightfully noticed by Mosses [33], such conventions are not only adhoc and language specific, but may also lead to erroneous definitions. Mosses' Modular SOS [33] (MSOS) brings modularity to SOS in a formal and elegant way, by grouping the non-syntactic configuration items into transition labels, and allowing rules to mention only those items of interest from each label. As discussed in Section 3.4, MSOS still inherits all the remaining limitations of SOS.

5. *Supports non-determinism and concurrency, at any desired granularity?*

   By inherently enforcing an interleaving semantics for concurrency, existing reduction semantics definitions (including ones based on evaluation contexts) can only capture a projection of

concurrency (when one's goal is to define a truly concurrent language), namely its resulting non-determinism. Proponents of existing reduction semantics approaches may argue that the resulting non-deterministic behavior of a concurrent system is all what matters, while proponents of true concurrency may argue that a framework which does not support naturally concurrent actions, i.e., actions that take place *at the same time*, is not a framework for concurrency. We do not intend to discuss the (admittedly important but debatable) distinctions between non-determinism and interleaving vs. true concurrency here. The fact that there are language designers who desire an interleaving semantics while others who desire a true concurrency semantics for their language is strong evidence that an ideal language definitional framework should simply support both, preferably with no additional settings of the framework, but rather via particular definitional methodologies within the framework.

6. *Is generic, that is, not tied to any particular programming language or paradigm?*

   A non-generic framework, i.e., one building upon a particular programming language or paradigm, may be hard or impossible to use at its full strength when defining a language that crosses the boundaries of the underlying language or paradigm. For example, a framework enforcing object or thread communication via explicit send and receive messages may require artificial encodings of languages that opt for a different communication approach (e.g., shared memory), while a framework enforcing static typing of programs in the defined language may be inconvenient for defining dynamically typed or untyped languages. In general, a framework providing and enforcing particular ways to define certain types of language features would lack genericity. Within an ideal framework, one can and should develop and adopt methodologies for defining certain types of languages or language features, but these should not be enforced. This genericity requirement is derived from the observation that today's programming languages are so diverse and based on orthogonal, sometimes even conflicting paradigms, that, regardless of how much we believe in the superiority of a particular language paradigm, be it object-oriented, functional or logical, a commitment to any existing paradigm would significantly diminish the strengths of a language definitional framework.

7. *Is executable, so one can "test" language or formal analyzer definitions, as if one already had an interpreter or a compiler for one's language? Efficient executability of language definitions may even eliminate the need for interpreters or compilers.*

   Most existing language definitional frameworks are, or until relatively recently were, lacking tool support for executability. Without the capability to execute language definitions, it is virtually impossible to debug or develop large and complex language definitions in a reasonable period of time. The common practice today is still to have a paper definition of a language using one's desired formalism, and *then* to implement an interpreter for the defined language following in principle the paper definition. This approach, besides the inconvenience of having to define the language twice, guarantees little to nothing about the appropriateness of the formal, paper definition. Compare this approach to an approach where there is *no gap* between the formal definition and its implementation as an interpreter. While any definition is by definition correct, one gets significantly more confidence in the appropriateness of a language definition, and is less reluctant to change it, when one is able to run it *as is* on tens or hundreds of programs. Recently, executability engines have been proposed both for MSOS (the MSOS tool, implemented by Braga and collaborators in Maude [7]) and for reduction semantics with evaluation contexts (the PLT Redex tool, implemented by Findler and his collaborators in

Scheme [21]). A framework providing *efficient* support for executability of formal language definitions may eliminate entirely the need to implement interpreters, or type checkers or type inferencers, for a language, because one can use directly the formal definition for that purpose.

8. *Has state-exploration capabilities, including exhaustive behavior analysis (e.g., finite-state model-checking), when one's language is non-deterministic or/and concurrent?*

   While executability of language definitions is indispensable when designing non-trivial languages, one needs richer tool support when the language is concurrent. Indeed, it may be that one's definition is appropriate for particular thread or process interleavings (e.g., when blocks are executed atomically), but that it has unexpected behaviors for other interleavings. Moreover, somewhat related to the desired computational granularity of language constructs mentioned in item 3 above, one may wish to exhaustively investigate all possible interleavings or executions of a particular concurrent program, to make sure that no undesired behaviors are present and no desired behaviors are excluded. When the state space of the analyzed program is large, manual analysis of behaviors may not be feasible; therefore, model-checking and/or safety property analysis are also desirable as intrinsic components of an ideal language definitional framework.

9. *Has a corresponding initial-model (to allow inductive proofs) or axiomatic semantics (to allow Hoare-style proofs), so that one can formally reason about programs?*

   To prove properties about programs in a defined programming language, or properties about the programming language itself, as also mentioned in item 2 above, the current practice is to encode/redefine the language semantics in a "richer" framework, such as a theorem prover, and then carry out the desired proofs there. Redefining the semantics of a fully fledged programming language in a different formalism is a highly nontrivial, error prone and tedious task, possibly taking months; automated translations may be possible when the original definition of the language is itself formal, though one would need to validate the translator. In addition to the "formal gap" mentioned in item 2 due to the translation itself, this process of redefining the language is simply inconvenient. An ideal language definitional framework should allow one to have, for each language, "one definition serving all purposes", including all those mentioned above.

   Current program verification approaches are based on axiomatic semantics (in the style of Hoare logic) of the language under consideration and on implementations of it in program verifiers. In fact, implementing program verifiers is still an art, one that few master. Existing program verifiers are based "in principle" on some implicit axiomatic semantics which is hand-crafted in the prover; a formal semantics is in fact not required and typically not given at all, thus creating an obvious gap between the implementation of a program verifier and the language semantics. Moreover, since axiomatic semantics are not executable and thus not testable, the underlying axiomatic semantics can be itself untrustable; at minimum, an alternative executable semantics of the language is needed and a proof that the axiomatic semantics is sound for it. Thus there is a double gap between a language definition and a program verifier for it. The very fact that one needs various semantics of a language for various purposes shows that none of these semantics is "ideal": as already stated above, an ideal language semantic definition should serve all the purposes.

There are additional desirable, yet of a more subjective nature and thus harder to quantify, requirements of an ideal language definitional framework. For example, it should be simple and easy to understand, teach and use by mainstream enthusiastic language designers, not only by language experts —in particular, an ideal framework should not require its users to have advanced concepts of category theory, logics, or type theory, in order to use it. Also, it should have good data representation capabilities and should allow proofs of theorems about programming languages that are easy to comprehend. Additionally, a framework providing support for parsing programs directly in the desired language syntax may be desirable to one requiring the implementation of an additional, external to the definitional setting, parser.

The nine requirements above are nevertheless ambitious. Some proponents of existing language definitional frameworks may argue that their favorite framework has these properties; however, a careful analysis of existing language definitional frameworks reveals that they actually fail to satisfy some of these ideal features (we discussed several such frameworks and their limitations in Chapter 3). Others may argue that their favorite framework has some of the properties above, the "important ones", declaring the other properties either "not interesting" or "something else". For example, one may say that what is important in one's framework is to get a dynamic semantics of a language, but its (model-theoretical) algebraic denotational semantics, proving properties about programs, model checking, etc., are "something else" and therefore are allowed to need a different "encoding" of the language. Our position is that an ideal language definitional framework should not compromise any of the nine requirements above.

Whether K satisfies all the requirements above or not is, and probably will always be, open. What we can mention with regards to this aspect, though, is that K was motivated and stimulated by the observation that the existing language definitional frameworks fail to fully satisfy these minimal requirements; consequently, K's design and development was conducted aiming *explicitly* to fulfill all nine requirements discussed above.

5.2 Motivating Example: IMP and IMP++ in K

We start by showing how the IMP and IMP++ languages, discussed in Chapter 3 in the context of the various seamantic approaches, can be defined using K. We define both a K semantics and a K type system for these languages; the later is defined mainly for demonstration reasons, but also since the former is agnostic to types, so one can execute even ill-formed programs.

5.2.1 K Semantics of IMP and IMP++

Recall from Section 3.9 that, in spite of its simplicity, IMP++ revealed limitations in each of the conventional semantic approaches; for example, big-step and small-step SOSs as well as denotational semantics were heavily non-modular, modular SOS required artificial syntactic extensions of the language in order to attain modularity, context reduction lacked modularity in some cases, the CHAM relied on a heavy airlock operation to match information in solution molecules, and all of the above lacked support for true concurrency (the CHAM provided more concurrency support than the others, but it still unnecessarily enforced interleaving in some cases). In this section we show that, at least in the context of the simple IMP and IMP++ languages, K avoids all these limitations.

Like in context reduction, K allows one to define evaluation contexts over the language syntax. However, unlike in context reduction, parsing does not play any crucial role in K, because K replaces the hard-to-implement split/plug operations of context reduction by plain, context-insensitive

Original language syntax	K Strictness	K Semantics
$AExp ::= Int \mid Var$		$\langle \underline{x} \cdots \rangle_k \ \langle \cdots x \mapsto i \cdots \rangle_{state}$ $\underline{i}$
$\mid AExp$ `+` $AExp$	$[strict]$	i_1 `+` $i_2 \rightarrow i_1 +_{Int} i_2$
$\mid AExp$ `/` $AExp$	$[strict]$	i_1 `/` $i_2 \rightarrow i_1 /_{Int} i_2$ where $i_1 \neq 0$
$BExp ::= AExp$ `<=` $AExp$	$[seqstrict]$	i_1 `<=` $i_2 \rightarrow i_1 \leq_{Int} i_2$
$\mid$ `not` $BExp$	$[strict]$	`not` $t \rightarrow \neg_{Bool} t$
$\mid BExp$ `and` $BExp$	$[strict(1)]$	`true and` $b \rightarrow b$ `false and` $b \rightarrow$ `false`
$Stmt ::=$ `skip`		`skip` $\rightarrow \cdot$
$\mid Var$ `:=` $AExp$	$[strict(2)]$	$\langle \underline{x := i} \cdots \rangle_k \ \langle \underline{\quad \sigma \quad} \rangle_{state}$ $\cdot \qquad \sigma[i/x]$
$\mid Stmt$ `;` $Stmt$		$s_1 ; s_2 \curvearrowright s_1 \curvearrowright s_2$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict(1)]$	`if true then` s_1 `else` $s_2 \ \rightarrow \ s_1$ `if false then` s_1 `else` $s_2 \ \rightarrow \ s_2$
$\mid$ `while` $BExp$ `do` $Stmt$		$\langle$ `while` b `do` s / `if` b `then` $(s$ `;` `while` b `do` $s)$ `else` $\cdot$ $\rangle_k$

Figure 5.1: K definition of IMP: syntax (left), syntax annotations (middle) and semantics (right) ($x \in Var$, $i, i_1, i_2 \in Int$, $t \in Bool$, $b \in BExp$, $s, s_1, s_2 \in Stmt$; the variables b, s, s_1, s_2 can also be in K)

rewriting. Therefore, instead of defining evaluation contexts using context-free grammars and relying on splitting syntactic terms (via parsing) into evaluation contexts and redexes, in K we define evaluation contexts using special rewrite rules. For example, the evaluation contexts of sum, comparison and conditional in IMP can be defined as follows (recall that sum was non-detereministic):

$$\begin{array}{rcl} a_1 \texttt{ + } a_2 & \rightleftharpoons & a_1 \curvearrowright \square \texttt{ + } a_2 \\ a_1 \texttt{ + } a_2 & \rightleftharpoons & a_2 \curvearrowright a_1 \texttt{ + } \square \\ a_1 \texttt{ <= } a_2 & \rightleftharpoons & a_1 \curvearrowright \square \texttt{ <= } a_2 \\ i_1 \texttt{ <= } a_2 & \rightleftharpoons & a_2 \curvearrowright i_1 \texttt{ + } \square \\ \texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2 & \rightleftharpoons & b \curvearrowright \texttt{if } \square \texttt{ then } s_1 \texttt{ else } s_2 \end{array}$$

The symbol $\rightleftharpoons$ stands for two special rewrite rules, one left-to-right and another right-to-left. The right-hand sides of the rules above contain two special operators: a *sequentialization* operator "$\curvearrowright$" read "and then" and *freezer* operators of the form "$\square$ `+` _". The first rule above says that in any expression of the form a_1 `+` a_2, a_1 can be scheduled for processing while a_2 is being held for future processing. Since the rules above are bi-directional, they can be used at will to structurally re-arrange the expressions to evaluate. Thus, when iteratively applied left-to-right they fulfill the role of *splitting* syntax into an evaluation context (the tail of the resulting sequence) and a redex (the head of the resulting sequence), and when applied right-to-left they fulfill the role of *plugging* syntax into context. Such structural re-arrangement rules are called *thermal rules* in K, sometimes also called *heating/cooling rules*, because they are reminiscent of the CHAM heating/cooling rules.

To avoid writing obvious thermal rules like the above, we prefer to use the *strictness attribute*

syntax annotations in K, as shown in the middle column in Figure 5.1: "*strict*" means non-deterministically strict in all enlisted arguments (given by their positions) or by default in all arguments if none enlisted, and "*seqstrict*" is like *strict* but each argument is fully processed before moving to the next one (see second thermal rule of "`<=`" above).

K takes a very abstract view of language syntax and it is not concerned *at all* with parsing aspects. More precisely, in K there is only one sort associated to all the language syntax, called K and staying for *computational structures* or *computations*, and terms t of sort K have the abstract syntax tree (AST) representation $l(t_1, ..., t_n)$, where l is some label and t_1,...,t_n are terms of sort K, extended with the list (infix) construct "$\curvearrowright$" (i.e., if t_1, ..., t_n are terms of sort K, then $t_1 \curvearrowright \cdots \curvearrowright t_n$ is also a term of sort K). Every original language construct, including constants and program variables, as well as all the freezer operations, are regarded as labels. For notational convenience, we continue to write K-terms using the original syntax instead of the harder to read AST notation.

Like in the CHAM, program or system configurations in K are organized as potentially nested structures of *cells* (we call them cells instead of molecules to avoid confusion with terminology in CHAM and chemistry). However, unlike the CHAM, K allows both multiset (or bag) and sequential (or list) cells, and K's cells may be labelled to distinguish them from each other. We use angle brackets as cell wrappers. The K configuration of IMP/IMP++ can be defined as follows:

$$\mathit{Config} \quad ::= \quad \mathsf{Bag}^{\cdot}_{\sqcup}[\langle K \rangle_k \mid \langle \mathsf{Map}^{\cdot}_{,}[\mathit{Var} \mapsto \mathit{Int}] \rangle_{\mathit{state}}]$$

Recall from Section 2.6 that the subscript and the superscript of the List, Bag, Set, or Map keyword, e.g., $\mathsf{Bag}^{\mathit{superscript}}_{\mathit{subscript}}[...]$, represents the infix construct and the unit of the corresponding structure (a set of bindings in the case of Map), respectively, and that the *binder* between the domain and the codomain of maps in $\mathsf{Map}^{\mathit{superscript}}_{\mathit{subscript}}[\mathit{domain}\ \ \mathit{binder}\ \mathit{codomain}]$, typically the symbol "$\mapsto$", represents the corresponding infix binding construct. For example, "$\langle$`x := 1; y := x+1`$\rangle_k\ \langle\cdot\rangle_{\mathit{state}}$" is a configuration holding program "`x := 1; y := x+1`" and empty state, and "$\langle$`x := 1; y := x+1`$\rangle_k\ \langle$`x` $\mapsto 0$, `y` $\mapsto 1\rangle_{\mathit{state}}$" is a configuration holding the same program and a state mapping `x` to 0 and `y` to 1. When we add threads (in IMP++), the configurations can hold multiple $\langle ... \rangle_k$ cells (its bag structure allows that).

The right column in Figure 5.1 shows the semantic K rules of IMP. A K rule is a term, called the *local context* of the rule, together with one or more changes in it; each change is given by an underlined subterm of the local context, which is the subterm to be replaced, together with a term underneath the line, which is its replacement. All changes in a K rule are applied in one parallel step. When the remaining components of a cell are not relevant, we use "$\cdots$" to match them. For example, the rule for variable lookup (first one) in Figure 5.1 says that if the first item in the computation cell is a variable $x \in \mathit{Var}$ and if the state cell contains a mapping $x \mapsto i$, then the x in the computation is replaced by i. The "$\cdots$" are *volatile*, in that other K rules may concurrently change data residing under the "$\cdots$". Ordinary rewrite rules are a special case of K rules, when the entire term is replaced; in this case, we typically use the standard notation $t \to u$ as syntactic sugar. K rules cannot always be desugared into rewrite rules (by combining all the changes into one top-level change), because, in addition to non-serializability aspects on K discussed in Section 5.3, by doing so one enforces interleaving in cases where K would be concurrent. For example, when we add threads to IMP in IMP++ below, various threads may lookup the same shared variable x concurrently with the K semantics based on K rules, while in a plain rewrite-based approach the various lookups would need to interleave, because the rule instances overlap on the state and in particular on the subterm $x \mapsto i$ in the state.

Let us discuss some of the non-obvious rules in Figure 5.1. The rule for `skip` simply disolves the `skip` statement. Indeed, the sort K of computations has a list structure, with "$\curvearrowright$" composing

Original language syntax	K Strictness	K Semantics
$AExp$ ::= $\ldots \mid$ `++` Var		$\langle \underline{\texttt{++}\,x}\ \cdots\rangle_k\ \langle \cdots\ x \mapsto \underline{i}\ \cdots\rangle_{state}$ $i +_{Int} 1 \qquad\qquad i +_{Int} 1$
$Stmt$ ::= $\ldots$ $\mid$ `output` $(AExp)$	$[strict]$	$\langle \underline{\texttt{output}\,(i)}\ \cdots\rangle_k\ \langle \cdots\ \underline{\cdot}\rangle_{output}$ $\cdot \qquad\qquad i$
$\mid$ `halt`		$\langle \underline{\texttt{halt}\cdots}\rangle_k$ $\cdot$
$\mid$ `spawn` $(Stmt)$		$\langle \underline{\texttt{spawn}\,(s)}\ \cdots\rangle_k\ \underline{\cdot}$ $\cdot \qquad\qquad \langle s\rangle_k$ $\langle\cdot\rangle_k \rightharpoonup \cdot$

Figure 5.2: K definition of IMP++ (extends that of IMP in Figure 5.1, *without changing anything*)

computations and with "$\cdot$" the empty computation ("$\cdot$" is the identity of the associative binary operation "$\curvearrowright$"). The rule for assignment performs two parallel changes, one disoving the assignment statement in the computation and one updating the state. The rule for sequential composition is a thermal rule, using "$\rightharpoonup$" instead of "$\rightarrow$", heating the "`;`" into "$\curvearrowright$". We prefer it to be a thermal instead of a normal rule because we do not want it to count as a computational step. Note that a reverse cooling rule is not necessary in this case. The rule for `while` captures the unrolling semantics, but note that the `while` statement must be the first item in the computation cell in order for the unrolling to be applied (otherwise the loop unrolling may continue forever).

Figure 5.2 shows how the K semantics of IMP can be seamlesly extended into a semantics for IMP++. To accomodate the output, a new cell needs to be added to the configuration:

$$Config \quad ::= \quad \mathsf{Bag}[\langle K\rangle_k \mid \langle \mathsf{Map}[Var \mapsto Int]\rangle_{state} \mid \langle \mathsf{List}[Int]\rangle_{output}]$$

However, note that none of the existing IMP rules needs to change, because each of them only matches what it needs from the configuration. The construct `output` is strict and its rule adds the value of its argument to the end of the output buffer (matches and replaces the unit "$\cdot$" at the end of the buffer). The rule for `halt` disolves the entire computation, and the rule for `spawn` creates a new $\langle ... \rangle_k$ cell wrapping the spawned statement. The code in this new cell will be processsed concurrently with the other threads. The last rule "cools" down a terminated thread by simply disolving it; it is a thermal rule because, again, we do not want it to count as a computation.

We conclude this section with a discusion on the concurrency of the K definition discussed above. As mentioned, since in K rule instances can share read-only data, various (actually all matching) instances of the look up rule can apply concurrently, in spite of the fact that they overlap on the state subterm. Similarly, since the rule for variable increment declares volatile everything else in the state except the mapping corresponding to the variable to increment, mutiple increments and reads of (distinct) variables can happen concurrently. However, if two threads want to increment the same variable, or if one wants to increment it while another wants to read it, then the two corresponding rules need to interleave, because the change of the increment rule overlaps with either the change or the read-only part of the other rule, so the two rule instances are in a concurrency conflict and cannot proceed at the same time. Note also that the rule for `output` matches and changes the end

of the output buffer; that means, in particular, that multiple outputs by various threads need to be interleaved for the same reason as above. On the other hand, the rule for `spawn` matches any empty top-level position and replaces it by the new thread, so multiple threads can spawn threads concurrently. Similarly, multiple threads can be disolved concurrently when they are done (last "cooling" thermal rule). Finally, note that our current rule for assignment matches and replaces the entire state, σ. That means that no other thread can read, write or increment any other variable concurrently with a variable assignment. This is of course undesirable and shows a (deliberate) poor concurrent language design. One would like instead to have a rule of the form:

$$\langle \underline{x \texttt{ := } i}_{\cdot} \cdots \rangle_k \ \langle \cdots \ x \mapsto \underline{\_}_{i} \cdots \rangle_{state}$$

(the underscore "_" is a nameless variable) This rule declares everything else in the state volatile as desired, but the problem is that it requires x to have already been previously assigned some value in the state. Thus, this rule can fully replace the one in Figure 5.1 only if we assume that programs are executed in states that already map all their variables. An alternative would be to keep both rules, so one can use the latter whenever the assigned variable is already in the state. A cleaner solution is given in Section 9, where one adds variable declarations to the language and one is allowed to only assign to already declared variables.

5.2.2 K Type Systems for IMP and IMP++

As discussed in Section 5.2.1, K is completely agnostic to typing. It may therefore be possible that the K semantics in Section 5.2.1 is applied even on ill-formed IMP/IMP++ programs, which may be considered undesirable by some language designers. On the other hand, the typing policy of this language is so obvious and simple, that any ordinary context-free parser can serve as a type checker. Nevertheless, since in K we prefer not to rely on parsing, and also for demonstration purposes, in this section we show how to define a type system using the very same K framework which, when executed, becomes a type checker. The idea is to define the type system also as an (executable) semantics of the language, but one in the more abstract domain of types rather than in the concrete domain of integer and boolean values. The technique is general and has been used to define more complex type systems, such as higher-order polymorphic ones (see Section 6.0.22).

To type IMP/IMP++ we only need one cell in the configuration, the one that holds the code to type: $\mathit{Config} ::= \langle K \rangle_k$. Figure 5.3 shows the IMP/IMP++ type system as a K system over such configurations. Constants reduce to their types, and types are propagated through each language construct in a straightforward manner. Note that almost each language construct is strict now, because we want to type all its arguments in almost all cases in order to apply the typing policy of the construct. Two constructs make exception, namely the increment and the assignment. The typing policy of these constructs is that they take precisely a variable and not something that types to an integer. If we defined, e.g., the assignment strict and with rule int `:=` int, then our type system would allow ill-formed programs like "`x+y := 0`". This is the reason for which we added a cell in the configuration and defined the typing of variables like in Figure 5.3, instead of naively with $x \to \mathit{int}$.

Original language syntax	K Strictness	K Semantics
$AExp ::= Int$		$i \to int$
$\mid Var$		$\langle \underline{x}_{int} \cdots \rangle_k$
$\mid AExp$ `+` $AExp$	$[strict]$	int `+` $int \to int$
$\mid AExp$ `/` $AExp$	$[strict]$	int `/` $int \to int$
$\mid$ `++` Var		`++` $x \to int$
$BExp ::= AExp$ `<=` $AExp$	$[strict]$	int `<=` $int \to bool$
$\mid$ `not` $BExp$	$[strict]$	`not` $bool \to bool$
$\mid BExp$ `and` $BExp$	$[strict]$	$bool$ `and` $bool \to bool$
$Stmt ::=$ `skip`		`skip` $\to stmt$
$\mid Var$ `:=` $AExp$	$[strict(2)]$	x `:=` $int \to stmt$
$\mid Stmt$ `;` $Stmt$	$[strict]$	$stmt$ `;` $stmt \to stmt$
$\mid$ `if` $BExp$ `then` $Stmt$ `else` $Stmt$	$[strict]$	`if` $bool$ `then` $stmt$ `else` $stmt \to stmt$
$\mid$ `while` $BExp$ `do` $Stmt$	$[strict]$	`while` $bool$ `do` $stmt \to stmt$
$\mid$ `output` `(` $AExp$ `)`	$[strict]$	`output` $int \to stmt$
$\mid$ `halt`		`halt` $\to stmt$
$\mid$ `spawn` `(` $Stmt$ `)`	$[strict]$	`spawn` `(` $stmt$ `)` $\to stmt$

Figure 5.3: K type system for IMP++ (and IMP)